The Journal of Logic and Algebraic Programming 80 (2011) 362–375

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Normal forms in total correctness for while programs and action systems

Kim Solin

Uppsala Universitet, Uppsala, Sweden

ARTICLE INFO

Article history:
Available online 13 April 2011

ABSTRACT

A classical while-program normal-form theorem is derived in demonic refinement algebra. In contrast to Kozen's partial-correctness proof of the theorem in Kleene algebra with tests, the derivation in demonic refinement algebra provides a proof that the theorem holds in total correctness. A normal form for action systems is also discussed.

1. Introduction

A classical folk theorem says that any while program can be simulated by a while program consisting of at most one loop, provided extra Boolean variables are allowed. The normal-form theorem for while loops was first published by Böhm and Jacopini [3], but according to Harel [7] this theorem was known to Kleene before that. Kozen [10] – based on a proof of Mirkowska [13] – showed how this theorem can be perspicuously proved in Kleene algebra with tests by elegant calculational derivations, and was the first to prove the normal form theorem without introducing an explicit assignment mechanism. However, Kleene algebra with tests only provides partial-correctness proofs. In this paper, we show how to obtain a *total-correctness* normal-form theorem. Our novel total-correctness proof is based on that of Kozen, but differs in that *refinement algebra* is used in the proof. Refinement algebras are abstract algebras intended for reasoning about program refinement in a *total-correctness setting*; the creator of the first such algebra is von Wright [16,17].

Using abstract algebra for reasoning about programs comes with several advantages. It is an approach that provides solid mathematical reasoning and at the same time reasoning which is simple and perspicuous; see for example [4,5,10–12,15–17] with references. The fact that an abstract algebra intended for reasoning about one structure can be modified only slightly to facilitate reasoning about another structure means that also proofs and definitions can be reused or only slightly modified. Our proof of the normal-form theorem in total correctness as based on Kozen's proof is a very good example of this. Höfner and Struth have also shown that abstract algebra presents a suitable level of abstraction when automation is concerned [8].

Inspired by the above-mentioned theorem for while program we also discuss a normal form for action systems and outline a proof that every action system can be put in normal form. The action–system formalism can be used for reasoning about concurrent systems [1].

We proceed as follows. First, we present demonic refinement algebra and its use for reasoning about programs. Then we consider commutativity conditions and a preservation technique, upon which we prove the normal-form theorem. Before concluding, we discuss a normal form for action systems. [1]

2. Demonic refinement algebra

The demonic refinement algebra of von Wright [16] is axiomatised over four operators and two constants. The first two operators, denoted ; and $\sqcap$, respectively, are binary infix and the last two, denoted * and $^\omega$ are unary postfix. The constants are denoted 1 and $\top$.

E-mail address: kim.solin@filosofi.uu.se

[1] The paper at hand was invited to this Special Issue and is an extension of an earlier version [14].

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.04.008

The intended intuition behind the operators and the constants is as follows. First of all, the carrier set of the algebra is to be seen as consisting of *programs* possibly containing demonic nondeterminism. This means that the operators are operators *on* programs and the constants are special programs. The *demonic-choice* operator $\sqcap$ applied to two programs, $x \sqcap y$, should be seen as a choice between x and y made by a demon.[2] That the choice is made by a demon means that we have no influence over it and that it can be done in the, for us, most undesirable way: striving to abortion. We will extensively use this way of looking at demonic choice in the sequel. The operator ; is *sequential composition*. It denotes sequential composition of programs: if x and y are programs, then $x; y$ denotes a program where, first, x is executed and, then, y is executed. The *weak-iteration* operator * is an iteration of any length that does terminate, whereas the *strong-iteration* operator $^\omega$ is an iteration that either terminates or goes on infinitely – which means abortion. The special program denoted by the constant $\top$ is the fictitious program magic that can establish any postcondition, and the special program denoted by 1 is skip, the immediately terminating program. Our terminology will be in the vein of Back and von Wright [2] and we will, for example, talk about execution of magic, although it is fictitious and cannot be implemented.[3]

We can now formulate the basic refinement algebra, which we call *demonic refinement algebra*.

Definition 2.1 [16]**.** A *demonic refinement algebra* (dRA) is a structure over the signature

$$(\sqcap, ;, ^*, ^\omega, \top, 1)$$

satisfying the following axioms and rules ($\sqcap$ has weakest precedence, followed by ;, and then * and $^\omega$, which have equal precedence – we omit ; so that $x; y$ is written xy when no confusion can arise):

$$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z, \tag{1}$$
$$x \sqcap y = y \sqcap x, \tag{2}$$
$$x \sqcap \top = x, \tag{3}$$
$$x \sqcap x = x, \tag{4}$$
$$x(yz) = (xy)z, \tag{5}$$
$$1x = x = x1, \tag{6}$$
$$\top x = \top, \tag{7}$$
$$x(y \sqcap z) = xy \sqcap xz, \tag{8}$$
$$(x \sqcap y)z = xz \sqcap yz, \tag{9}$$
$$x^* = 1 \sqcap xx^*, \tag{10}$$
$$x \sqsubseteq yx \sqcap z \Rightarrow x \sqsubseteq y^*z, \tag{11}$$
$$x \sqsubseteq xy \sqcap z \Rightarrow x \sqsubseteq zy^*, \tag{12}$$
$$x^\omega = 1 \sqcap xx^\omega, \tag{13}$$
$$yx \sqcap z \sqsubseteq x \Rightarrow y^\omega z \sqsubseteq x \text{ and} \tag{14}$$
$$x^\omega = x^* \sqcap x^\omega \top, \tag{15}$$

where the order $\sqsubseteq$ is defined by $x \sqsubseteq y \Leftrightarrow_{df} x \sqcap y = x$. ◁

The *refinement ordering* $\sqsubseteq$ on the algebra defined above is to be read "y establishes anything that x does and possibly more" (intuitively, if x is refined by y, then a demon would always choose x since y can do anything that x does and possibly more; by choosing x the demon has a better chance of winning).

It can be shown that all the operators are isotone with respect to the refinement ordering $\sqsubseteq$ and that $\sqsubseteq$ is a partial order. The reduct structure over the signature $(\sqcap, ;, \top, 1)$ is an idempotent semiring, and the reduct structure over the signature $(\sqcap)$ is a bounded greatest-lower-bound semilattice, with $\top$ as the greatest element. In comparison to Kleene algebra [9], the axiom preventing reasoning about total correctness ($x\top = \top$) has been removed and strong iteration has been added.

We define a syntactic constant $\bot$ with the intuition that it stands for an always nonterminating program, an abort statement [16]:

$$\bot =_{df} 1^\omega.$$

[2] The intuition is based on the game-theoretic view on programs presented in Chapter 14 of Back and von Wright [2]. The notion "demon" is of course only to be understood metaphorically.

[3] Some would object that talking about execution of magic is nonsense, since it is nonsensical to talk about execution of a nonimplementable program: a program that cannot be implemented is no program at all and can certainly not be executed. Some would not. (The idea of using magic was introduced, independently of each other, by C.C. Morgan, J.M. Morris, and G. Nelson in the 1980s.)

We thus equate abortion and (idle) nontermination. The syntactic constant $\bot$ is a least element and a left annihilator, as the following proposition states.

Proposition 1 [16]. *Let x be an element in the carrier set of a* dRA. *Then*

$$\bot \sqsubseteq x \text{ and} \tag{16}$$

$$\bot x = \bot \tag{17}$$

hold.

Axioms will usually be referred to by number, but for convenience the properties will sometimes only be referred to by their canonical name, such as *associativity*, *commutativity*, *idempotence*, *skip* or *annihilation*. Axioms (8) and (9) will be referred to as *distributivity* and axioms (10) and (13) will be referred to as *unfolding*; axioms (11), (12) and (14) as *induction*; and axiom (15) as *isolation*.

Let us look at the program-theoretic intuition behind some of the axioms. The third axiom says that a demon choosing between a miracle and a program x will always choose x. This is because the demon always wants to establish abortion, and if magic is executed then this is not possible. The seventh axiom says that after magic has been executed nothing affects the program any more. The tenth axiom says that a finite iteration can be seen as an unfolding of the iterated statement: x is repeated any finite number of times until, finally, 1 is chosen, the program skips and, so, the iteration ends. Axiom (11) says that if $x \sqsubseteq yx \sqcap z$, then x can be refined by a succession of ys, ending with z:

$$x \sqsubseteq yx \sqsubseteq yyx \sqsubseteq yyyx \sqsubseteq \cdots \sqsubseteq yyy \cdots z.$$

That is, y can be repeated any finite number of times and then followed by z, in other words y^*z. Axiom (12) is analogous. Axiom (14) says the same thing as axiom (11), but now the iteration might possibly not terminate. The reason we do not have a strong-iteration axiom analogous to (12) is related to our ability to express magic: Take for example $x = y = 1$ and $z = \top$. Then the left-hand side of a strong-iteration induction rule analogous to (12) would hold, whereas the right-hand side would not. Axiom (15) separates an iteration into its finite and infinite parts: the finite part is given by weak iteration and the infinite part is given by the $x^\omega\top$. The intuition behind $x^\omega\top$ denoting the infinite part is that unless the iteration goes on forever, and thus aborts, a demon would not choose that alternative, since this would result in a miracle. The remaining axioms can easily be given similar interpretations.

So spoke the intuition. But one could argue that this concordance of the axioms with the every-day understanding of program-theoretic constructs does not actually justify anything. It is yet to be shown how this algebra mathematically relates to how one traditionally has understood programs formally. One, but not necessarily the only, such relation is given by the following fact: the set of conjunctive predicate transformers over a fixed state space equipped with standard operators (matching the above interpretation in a predicate-transformer setting) [2,6] forms a demonic refinement algebra [16,17].

Formally, a conjunctive predicate transformer is a function

$$S : \wp(\Sigma) \to \wp(\Sigma),$$

such that for a nonempty I, it satisfies

$$S.\Big(\bigcap_{i\in I} q_i\Big) = \bigcap_{i\in I} S.q_i,$$

where Σ is any set. Let CTran_Σ denote the set of conjunctive predicate transformers over Σ. If one defines the following operators for any $q \in \wp(\Sigma)$

$$\begin{aligned}(S; T).q &=_{df} S.T.q,\\ (S \sqcap T).q &=_{df} S.q \cap T.q,\\ S^* &=_{df} \nu.(\lambda X \bullet S; X \sqcap \mathsf{skip}) \text{ and}\\ S^\omega &=_{df} \mu.(\lambda X \bullet S; X \sqcap \mathsf{skip}),\end{aligned}$$

where μ denotes the least fixpoint and ν the greatest, and the constants

$$\begin{aligned}\mathsf{abort} &=_{df} (\lambda q \bullet \emptyset),\\ \mathsf{magic} &=_{df} (\lambda q \bullet \Sigma) \text{ and}\\ \mathsf{skip} &=_{df} (\lambda q \bullet q),\end{aligned}$$

then the structure

$$(\mathsf{CTran}_\Sigma, \sqcap, ;, , ^*, ^\omega, \mathsf{magic}, \mathsf{skip})$$

is a dRA [16,17]. The existence of the fixpoints is ensured by the fact that conjunctive predicate transformers are monotone, closed under composition and, by point-wise extension, form a complete lattice, so *eo ipso* Knaster-Tarski's theorem applies.

Although an important and certainly the motivating model for refinement algebra, the predicate transformer model is not to be seen as excluding other interpretations. Moreover, in our view, the most important understanding of the algebra lies not in this so-called semantics, but in the informal exposition of the axioms.

The *leapfrog* and *decomposition* properties

$$x(yx)^\omega = (xy)^\omega x \text{ and} \tag{18}$$

$$(x \sqcap y)^\omega = x^\omega (yx^\omega)^\omega, \tag{19}$$

respectively, have been proved by von Wright [16] and will be used later on.

3. Guards and assertions

An element g of the carrier set that has a complement $\bar{g}$ satisfying

$$g\bar{g} = \bar{g}g = \top \quad \text{and} \quad g \sqcap \bar{g} = 1 \tag{20}$$

is called a *guard*. We collectively refer to the guards as *the set of guards* and we will use the symbols f, g and h for denoting guards (if needed, indexed with natural numbers). Intuitively, guards are statements that check whether a predicate holds and, if so, skip, otherwise do magic. The first guard axiom says that either a predicate or its negation holds, so a sequential composition of a guard and its complement is always a miracle. The second guard axiom says that a demon will always be able to skip when choosing between a guard or the guard's complement.

As the following proposition states, the set of guards forms a Boolean algebra.

Proposition 2 [16]. *Let G be the set of guards of a* dRA. *Then*

$$(G, \sqcap, ;, \bar{}, 1, \top)$$

is a Boolean algebra, where $\sqcap$ *is meet,* $;$ *is join,* $\bar{}$ *is complement, 1 is the bottom element, and* $\top$ *is the top element.*

Every guard g is defined to have a corresponding *assertion*

$$g^\circ = \bar{g}\bot \sqcap 1. \tag{21}$$

This means that $^\circ$ is a mapping from guards to a subset of the carrier set, the *set of assertions*. Assertions are similar to guards, but abort if the predicate does not hold. If the predicate does not hold, then a demon would choose the left-hand side of the demonic choice and the negated guard would skip and the whole program abort (which is what a demon wants). If, on the other hand, the predicate holds, then a demon would choose the right-hand side, since otherwise the negated guard would do magic and the demon could then no longer establish abortion. Note that $\bar{g}^\circ$ means that the assertion operator $^\circ$ is applied to the guard $\bar{g}$ (and not to be read the other way around, that the complement operator is applied to the assertion g°). Assertions will not be used in this paper, but are mentioned to make the presentation of demonic refinement algebra complete. The abstract-algebraic guards and assertions have canonical interpretations as predicate transformers (see [16]).

The assertion-skip-guard property (the *asg property*) is an especially important property, which states that

$$g^\circ \sqsubseteq 1 \sqsubseteq h \tag{22}$$

holds for any guards g and h. The property is immediate from the definition of assertion and the fact that the guards form a Boolean algebra. We will also make use of the fact that

$$\bar{g}(gx)^\omega = \bar{g} \tag{23}$$

holds (by unfolding of strong iteration (13), the definition of guards and the fact that $\top$ is the top element (3)).

4. Conditionals and loops

To show how the algebra relates to traditional program constructs we here show how to do encodings of conditionals and while loops. This means that we can express the classical while language: sequential composition, conditional and iteration.

Conditionals are one of the basic building blocks of any programming language. A conditional checks whether a predicate holds, and depending on this it chooses between two actions. Traditionally, it is written

$$\mathsf{if}\ g\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi},$$

and should be understood so that if g holds, then x is executed, otherwise y is executed. In the algebra we can encode this as $gx \sqcap \bar{g}y$ with the rationale that the demon always chooses the statement for which the guard holds. If the guard does not hold, then the guard performs a miracle. If the demon, then, would choose to execute that statement the demon could never establish abortion. Note that the false predicate is thus represented by a guard that is always miraculous, that is, by $\top$.

Another central construct in any programming language is the while loop, or loop for short. A loop

$$\text{while } g \text{ do } x \text{ od}$$

iterates a program statement x any number of times as long as the predicate g holds. If the predicate always holds, then the loop will iterate infinitely – a usually undesirable scenario. In dRA we have two possibilities of modeling a loop: one using weak iteration, $(gx)^*\bar{g}$, and another using strong iteration, $(gx)^\omega\bar{g}$. If weak iteration is used to model the loop we are assuming that the iteration terminates. All that is proved about a loop encoded using weak iteration thus assumes that the iteration is terminating (nevertheless, the loop might still be nonterminating (aborting) if the iterated program statement is aborting). If, on the other hand, strong iteration is used, we do not need to assume that the loop is terminating – indeed, everything we prove about a loop using the strong iteration operator holds when the loop is terminating *as well as* when it is nonterminating.

5. Commutativity conditions

In order to state the theorem, a commutativity condition of the form

$$\begin{cases} gxg = gx \\ \bar{g}x\bar{g} = \bar{g}x \end{cases}, \text{ or equivalently, of the form } \begin{cases} xg \sqsubseteq gx \\ x\bar{g} \sqsubseteq \bar{g}x \end{cases},$$

must be made on the programs involved. Intuitively, the condition above says that "if the program x terminates, it preserves g." If the program aborts, anything can happen. We will say that x *preserves* g if x and g meet the above condition. The two equivalent conditions above correspond to Kozen's commutativity conditions, but since we want to prove total correctness, it makes sense not to assume termination of the programs. This means that, unlike Kozen, we cannot make assumptions of the form $gx \sqsubseteq xg$, since this would imply that x must terminate (cf. the total-correctness condition in [16,17]; in Kleene algebra, the characterisations of total and weak correctness coincide). Note that the first part (the first line) of the condition does not imply the second and vice versa (a concrete counterexample can be constructed).

To illustrate this technique, Kozen [10] uses

$$\text{if } g \text{ then } x; y_1 \text{ else } x; y_2 \text{ fi}$$

as an example of a conditional that is to be simplified. We now reuse this example and resettle the properties in total correctness. First assume that g is preserved by x, that is assume that

$$\begin{cases} gxg = gx \\ \bar{g}x\bar{g} = \bar{g}x \end{cases}$$

holds. Then the program can be rewritten into a more separated form as

$$x; \text{ if } g \text{ then } y_1 \text{ else } y_2 \text{ fi},$$

which can be formulated and proved in dRA by

$$\begin{aligned} & x(gy_1 \sqcap \bar{g}y_2) \\ = \ & \{\text{distributivity}\} \\ & xgy_1 \sqcap x\bar{g}y_2 \\ = \ & \{\text{axiom (6), definition of guards (20)}\} \\ & (g \sqcap \bar{g})xgy_1 \sqcap (g \sqcap \bar{g})x\bar{g}y_2 \\ = \ & \{\text{distributivity (9)}\} \\ & gxgy_1 \sqcap \bar{g}xgy_1 \sqcap gx\bar{g}y_2 \sqcap \bar{g}x\bar{g}y_2 \\ = \ & \{\text{preservation assumption}\} \\ & gxy_1 \sqcap \bar{g}x\bar{g}gy_1 \sqcap gxg\bar{g}y_2 \sqcap \bar{g}xy_2 \\ = \ & \{\text{definition of guards, axiom (7)}\} \end{aligned}$$

$$
\begin{aligned}
& gxy_1 \sqcap \bar{g}x\top \sqcap gx\top \sqcap \bar{g}xy_2 \\
= \; & \{\text{axiom (2), distributivity (8)}\} \\
& gx(y_1 \sqcap \top) \sqcap \bar{g}x(y_2 \sqcap \top) \\
= \; & \{\text{axiom (3)}\} \\
& gxy_1 \sqcap \bar{g}xy_2 .
\end{aligned}
$$

Although similar, the proof is different from Kozen's, since we work in total correctness and thus in demonic refinement algebra and with the above preservation conditions.

It is easy to show that if e is a well-formed expression consisting of elements from the carrier set and the operators ; and $\sqcap$ and all the carrier-set elements preserve a guard g, then the whole expression e preserves g. This, in turn, means that x^ω preserves g (this is the invariant rule for strong iteration in weak correctness). We will sometimes refer to this fact as "move guard."

Moreover, assuming that x preserves g, it can be shown – by induction, the assumption, distributivity, and unfolding – that

$$(gx)^\omega g \sqsubseteq gx^\omega \tag{24}$$

holds, a fact which we will employ later on.

6. Kozen's preservation technique

Suppose we would like to preserve the value of g across the program x, but we cannot assume that x preserves g. To do this we need to first introduce a new guard h and assume that x preserves h. Then we can set h to g by a special program $z; (g \leftrightarrow h)$, where $g \leftrightarrow h =_{df} hg \sqcap \bar{h}\bar{g}$, and this program can then be injected into an appropriate place. This corresponds, on an abstract level, to adding extra Boolean variables. As Kozen [10] notes, the intuition is that z assigns the value of g to some new Boolean variable that is tested by h. The guard $g \leftrightarrow h$ says that g and h have the same Boolean value just after execution of z [10]. This technique was used by Kozen [10] in his seminal paper on Kleene algebra with tests.

Consider again the example with the conditional from the previous section, but assume now that g is not preserved by x. Then we can use Kozen's technique to first "store" the value of g in a new guard h which is preserved by x, and then prove that

$$z; (g \leftrightarrow h); \text{if } g \text{ then } x; y_1 \text{ else } x; y_2 \text{ fi}$$

is equivalent to

$$z; (g \leftrightarrow h); x; \text{if } h \text{ then } y_1 \text{ else } y_2 \text{ fi} .$$

This is done as follows. Assume

$$\begin{cases} hxh = hx \\ \bar{h}x\bar{h} = \bar{h}x, \end{cases}$$

and derive

$$
\begin{aligned}
& (g \leftrightarrow h)(gxy_1 \sqcap \bar{g}xy_2) \\
= \; & \{\text{definition}\} \\
& (gh \sqcap \bar{g}\bar{h})(gxy_1 \sqcap \bar{g}xy_2) \\
= \; & \{\text{distributivity}\} \\
& ghgxy_1 \sqcap \bar{g}\bar{h}gxy_1 \sqcap gh\bar{g}xy_2 \sqcap \bar{g}\bar{h}\bar{g}xy_2 \\
= \; & \{\text{guards form a Boolean algebra}\} \\
& ghgxy_1 \sqcap \bar{g}g\bar{h}xy_1 \sqcap g\bar{g}hxy_2 \sqcap \bar{g}\bar{h}\bar{g}xy_2 \\
= \; & \{\text{definition of guards (20), axiom (7)}\} \\
& ghgxy_1 \sqcap \top \sqcap \top \sqcap \bar{g}\bar{h}\bar{g}xy_2 \\
= \; & \{\text{axiom (3)}\} \\
& ghgxy_1 \sqcap \bar{g}\bar{h}\bar{g}xy_2 \\
= \; & \{\text{guards form a Boolean algebra}\}
\end{aligned}
$$

$$
\begin{aligned}
& ghxy_1 \sqcap \bar{g}\bar{h}xy_2 \\
= \; & \{\text{axiom (3)}\} \\
& ghx(y_1 \sqcap \top) \sqcap \bar{g}\bar{h}x(y_2 \sqcap \top) \\
= \; & \{\text{preservation assumption, guards form a BA, distributivity}\} \\
& ghxhy_1 \sqcap ghx\bar{h}y_2 \sqcap \bar{g}\bar{h}xhy_1 \sqcap \bar{g}\bar{h}x\bar{h}y_2 \\
= \; & \{\text{distributivity}\} \\
& (gh \sqcap \bar{g}\bar{h})x(hy_1 \sqcap \bar{h}y_2) \\
= \; & \{\text{definition}\} \\
& (g \leftrightarrow h)x(hy_1 \sqcap \bar{h}y_2).
\end{aligned}
$$

By isotony, we have then proved the claim.

7. The normal form theorem

We will say that a while program is in *normal form* if it is of the form

$$x;\ \text{while}\, g\, \text{do}\, y\, \text{od},$$

where x and y do not contain while loops. Using Kleene algebra with tests Kozen [10] proved that every while program can be written in normal form. Kozen thus proved the theorem in partial correctness, but here we use refinement algebra to obtain a theorem in total correctness.

Theorem 7.1. *Every (possibly nonterminating) while program, appropriately augmented with subprograms of the form* z; $(g \leftrightarrow h)$ *and when reasoning under preservation assumptions of the form*

$$\begin{cases} gxg = gx \\ \bar{g}x\bar{g} = \bar{g}x \end{cases},$$

is equivalent to a while program in normal form.

Proof. The theorem is proved by induction on the structure of while programs. Following Kozen [10], who in turn follows Mirkowska [13], we give a method for moving an inner while loop to the outside for every program construct. That we are working in demonic refinement algebra and encode the loop with strong iteration in order to obtain total correctness means that several of the individual steps must be done quite differently from Kozen's.

Step 1: Conditional. Consider the program

$$\begin{aligned}
&\text{if } g \ \text{ then } x_1 \text{ while } f_1 \text{ do } y_1 \text{ od} \\
&\quad \text{else } x_2 \ \text{ while } f_2 \text{ do } y_2 \text{ od fi}\ .
\end{aligned}$$

To show how to move the while loops outside, we first introduce a new test h and program z that sets h to g. We also assume that h is preserved by the programs x_1, x_2, y_1 and y_2. Having taken on these assumptions, we prove that

$$\begin{aligned}
&z;\ (g \leftrightarrow h);\ \text{if } g \ \text{ then } x_1 \text{ while } f_1 \text{ do } y_1 \text{ od} \\
&\qquad\qquad\qquad\quad \text{else } x_2 \ \text{ while } f_2 \text{ do } y_2 \text{ od fi}
\end{aligned} \tag{25}$$

and

$$\begin{aligned}
&z;\ (g \leftrightarrow h);\ \text{if } h \text{ then } x_1 \text{ else } x_2 \text{ fi}\ ; \\
&\text{while } (hf_1 \sqcap \bar{h}f_2) \text{ do} \\
&\ \ \text{if } h \text{ then } y_1 \text{ else } y_2 \text{ fi} \\
&\text{od}
\end{aligned} \tag{26}$$

are equivalent. To prove that (25) and (26) are equivalent, the beginning z can be removed and the remaining parts be shown equivalent – this follows from isotony. Encoding into demonic refinement algebra and then using distributivity, the guard definition, axiom (7) and Boolean algebra for simplifying, the first expression (25) takes the form

$$ghx_1(f_1y_1)^\omega\bar{f}_1 \ \sqcap\ \bar{g}\bar{h}x_2(f_2y_2)^\omega\bar{f}_2. \tag{27}$$

Similarly, the second expression (26) becomes

$$(ghx_1 \sqcap \bar{g}\bar{h}x_2)(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega(h\bar{f_1} \sqcap \bar{h}\bar{f_2}). \tag{28}$$

To see that this is indeed so, use the basic equality

$$hf_1 \sqcap \bar{h}f_2 = (\bar{h} \sqcap f_1)(h \sqcap f_2),$$

de Morgan rules and double negation on the subexpression $\overline{hf_1 \sqcap \bar{h}f_2}$, and then distributivity on the remaining subexpression; this is exactly like in Kozen's paper [10]. The second expression (28) is in fact equivalent to

$$\begin{array}{l} \quad ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega h\bar{f_1} \\ \sqcap \\ \quad ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega \bar{h}\bar{f_2} \\ \sqcap \\ \quad \bar{g}\bar{h}x_2(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega h\bar{f_1} \\ \sqcap \\ \quad \bar{g}\bar{h}x_2(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega \bar{h}\bar{f_2} \end{array} \tag{29}$$

by distributivity (cf. again Kozen [10]). We now show how the equality of (27) and (29) can be derived.

For the reverse refinement, $\sqsupseteq$, it suffices to derive

$$\begin{array}{ll} & ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega h\bar{f_1} \\ \sqsubseteq & \{\text{isotony}\} \\ & ghx_1(hf_1y_1)^\omega h\bar{f_1} \\ \sqsubseteq & \{\text{property (24)}\} \\ & ghx_1h(hf_1y_1)^\omega \bar{f_1} \\ = & \{\text{preservation assumption}\} \\ & ghx_1(hf_1y_1)^\omega \bar{f_1}, \end{array}$$

that is, to derive the left-hand side (with respect to $\sqcap$) of (27) from the first part of (29). The right-hand side of (27) follows symmetrically from the fourth part of (29). By this and isotony we have shown the reverse refinement.

For the refinement, $\sqsubseteq$, we derive

$$\begin{array}{ll} & ghx_1(f_1y_1)^\omega \bar{f_1} \\ \sqsubseteq & \{\text{asg property (22)}\} \\ & ghx_1(hf_1y_1)^\omega h\bar{f_1} \\ = & \{\text{guards form a Boolean algebra, axiom (3)}\} \\ & ghx_1(hhf_1y_1 \sqcap \top)^\omega h\bar{f_1} \\ = & \{\text{guards form a Boolean algebra, axiom (7), distributivity}\} \\ & ghx_1(h(hf_1y_1 \sqcap \bar{h}f_2y_2))^\omega h\bar{f_1} \\ \sqsubseteq & \{\text{property (24)}\} \\ & ghx_1h(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega \bar{f_1} \\ = & \{\text{preservation assumption}\} \\ & ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega \bar{f_1} \\ \sqsubseteq & \{\text{asg property (22)}\} \\ & ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega h\bar{f_1}. \end{array}$$

We also derive

$$
\begin{aligned}
& ghx_1(f_1y_1)^\omega\bar{f}_1 \\
\sqsubseteq\ & \{\text{asg property (22), isotony}\} \\
& ghx_1(hf_1y_1)^\omega\top \\
=\ & \{\text{definition of guards (20), axiom (7)}\} \\
& ghx_1(hf_1y_1)^\omega h\bar{h}\bar{f}_2 \\
\sqsubseteq\ & \{\text{preservation assumption, move guard}\} \\
& ghx_1h(hf_1y_1)^\omega\bar{h}\bar{f}_2 \\
\sqsubseteq\ & \{\text{assumption A, see below}\} \\
& ghx_1h(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega\bar{h}\bar{f}_2 \\
=\ & \{\text{preservation assumption}\} \\
& ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega\bar{h}\bar{f}_2,
\end{aligned}
$$

where for any x and any guard g, assumption A is proved by

$$
\begin{aligned}
& (gx)^\omega g \sqsubseteq gx^\omega \\
\Leftarrow\ & \{\text{induction}\} \\
& gxgx^\omega \sqcap g \sqsubseteq gx^\omega \\
\Leftrightarrow\ & \{\text{preservation assumption, distributivity}\} \\
& g(xx^\omega \sqcap 1) \sqsubseteq gx^\omega \\
\Leftrightarrow\ & \{\text{unfold strong iteration}\} \\
& \mathsf{true}.
\end{aligned}
$$

By this, we have established that

$$ghx_1(f_1y_1)^\omega\bar{f}_1 \ \sqsubseteq\ ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega hf_1 \ \sqcap\ ghx_1(hf_1y_1 \sqcap \bar{h}f_2y_2)^\omega\bar{h}\bar{f}_2,$$

and symmetric reasoning can be used to show that the right-hand side of (27) derives the two remaining parts of (29).

Step 2: Nested loops. In this step, we can follow Kozen [10], since no commutativity conditions are needed – nevertheless, strong iteration is different from weak iteration and this must be taken into consideration.

We first show that

$$\mathsf{while}\ f\ \mathsf{do}\ x;\ \mathsf{while}\ g\ \mathsf{do}\ y\ \mathsf{od}\ \mathsf{od} \tag{30}$$

is equal to

$$
\begin{aligned}
&\mathsf{if}\ f \\
&\ \mathsf{then}\ x;\ \mathsf{while}\ f \sqcap g\ \mathsf{do}\ \mathsf{if}\ g\ \mathsf{then}\ y\ \mathsf{else}\ x\ \mathsf{fi}\ \mathsf{od} \\
&\ \mathsf{else}\ \ \mathsf{skip} \\
&\mathsf{fi}
\end{aligned}
\tag{31}
$$

and then the normal form follows from applying the rule in Step 1. It is easy to show that the skip clause in the conditional is equivalent to

$$\mathsf{skip};\ \mathsf{while}\ \top\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od},$$

so that the second form exactly matches that of Step 1.

Program (30) takes the form

$$(fx(gy)^\omega\bar{g})^\omega\bar{f}$$

in refinement algebra, and (31) becomes

$$fx(gy \sqcap \bar{g}fx)^\omega\bar{f}\bar{g} \sqcap \bar{f},$$

after simplification with distributivity and Boolean algebra. We now calculate

$$
\begin{array}{ll}
 & (fx(gy)^\omega \bar{g})^\omega \bar{f} = fx(gy \sqcap \bar{g}fx)^\omega \bar{f}\bar{g} \sqcap \bar{f} \\
\Leftarrow & \{\text{unfolding (13), isotony}\} \\
 & (gy)^\omega \bar{g}(fx(gy)^\omega \bar{g})^\omega \bar{f} = (gy \sqcap \bar{g}fx)^\omega \bar{f}\bar{g} \\
\Leftrightarrow & \{\text{leapfrog (18)}\} \\
 & (gy)^\omega (\bar{g}fx(gy)^\omega)^\omega \bar{g}\bar{f} = (gy \sqcap \bar{g}fx)^\omega \bar{f}\bar{g} \\
\Leftrightarrow & \{\text{guards form a Boolean algebra}\} \\
 & (gy)^\omega (\bar{g}fx(gy)^\omega)^\omega \bar{f}\bar{g} = (gy \sqcap \bar{g}fx)^\omega \bar{f}\bar{g} \\
\Leftrightarrow & \{\text{decomposition}\} \\
 & \text{true}
\end{array}
$$

and have so proved what we wanted to establish.

Step 3: Eliminating postcomputations. In this step of the proof we show that a computation that is to be executed after a while loop can be included in the while loop. More precisely, we first show that

$$\text{while } g \text{ do } x \text{ od}; \ y \tag{32}$$

is equal to

$$
\begin{array}{l}
\text{if } \bar{g} \text{ then } y \text{ else } \ \text{while } g \ \ \text{do } x; \\
\qquad\qquad\qquad\qquad\quad \text{if } \bar{g} \text{ then } y \text{ else skip fi} \\
\qquad\qquad\qquad\qquad\quad \text{od} \\
\text{fi}
\end{array} \tag{33}
$$

under the assumption that y preserves g. (The case where g is not preserved by y is dealt with later.)

If x and y are while free, then the else clause of (33) is in normal form and the whole program can thus be transformed into normal form by Step 1.

Expression (32) can be encoded into refinement algebra as $(gx)^\omega \bar{g}y$, which after unfolding the omega once by axiom (13) and then distributing becomes $gx(gx)^\omega \bar{g}y \sqcap \bar{g}y$. The second expression becomes $\bar{g}y \sqcap g(gx(\bar{g}y \sqcap g))^\omega \bar{g}$ after applying axiom (6). We now show that

$$(gx)^\omega \bar{g}y \sqsubseteq \bar{g}y \sqcap g(gx(\bar{g}y \sqcap g))^\omega \bar{g}$$

and

$$\bar{g}y \sqcap g(gx(\bar{g}y \sqcap g))^\omega \bar{g} \sqsubseteq gx(gx)^\omega \bar{g}y \sqcap \bar{g}y$$

and have thereby shown what we wanted.

For the first claim, it suffices to calculate

$$
\begin{array}{ll}
 & (gx)^\omega \bar{g}y \sqsubseteq \bar{g}y \sqcap g(gx(\bar{g}y \sqcap g))^\omega \bar{g} \\
\Leftrightarrow & \{\text{distributivity}\} \\
 & (gx)^\omega \bar{g}y \sqsubseteq \bar{g}y \sqcap g(gx\bar{g}y \sqcap gxg)^\omega \bar{g} \\
\Leftarrow & \{\text{induction (14)}\} \\
 & gx(\bar{g}y \sqcap g(gx\bar{g}y \sqcap gxg)^\omega \bar{g}) \sqcap \bar{g}y \sqsubseteq \bar{g}y \sqcap g(gx\bar{g}y \sqcap gxg)^\omega \bar{g} \\
\Leftarrow & \{\text{isotony}\} \\
 & gx(\bar{g}y \sqcap g(gx\bar{g}y \sqcap gxg)^\omega \bar{g}) \sqsubseteq g(gx\bar{g}y \sqcap gxg)^\omega \bar{g} \\
\Leftrightarrow & \{\text{short hand: } K =_{df} (gx\bar{g}y \sqcap gxg)^\omega \bar{g}, \text{ unfolding (13), distributivity, guards form a Boolean algebra}\} \\
 & gx(\bar{g}y \sqcap gK) \sqsubseteq gx\bar{g}yK \sqcap gxgK \sqcap g\bar{g} \\
\Leftrightarrow & \{\text{guards form a Boolean algebra, axiom (3)}\}
\end{array}
$$

$$
\begin{aligned}
& gx(\bar{g}y \sqcap gK) \sqsubseteq gx\bar{g}yK \sqcap gxgK \\
\Leftarrow\ & \{\text{distributivity, isotony}\} \\
& gx\bar{g}y \sqsubseteq gx\bar{g}yK \\
\Leftrightarrow\ & \{\text{preservation assumption}\} \\
& gx\bar{g}y \sqsubseteq gx\bar{g}y\bar{g}K \\
\Leftrightarrow\ & \{\text{distributivity, property (23)}\} \\
& gx\bar{g}y \sqsubseteq gx\bar{g}y\bar{g} \\
\Leftrightarrow\ & \{\text{preservation assumption}\} \\
& gx\bar{g}y \sqsubseteq gx\bar{g}y \\
\Leftrightarrow\ & \{\text{reflexivity of refinement}\} \\
& \mathsf{true}.
\end{aligned}
$$

For the second claim, we have that

$$
\begin{aligned}
& g(gx(\bar{g}y \sqcap g))^{\omega}\bar{g} \\
=\ & \{\text{unfolding (13), distributivity, guards form a BA, axiom (3)}\} \\
& gx(\bar{g}y \sqcap g)(gx(\bar{g}y \sqcap g))^{\omega}\bar{g} \\
=\ & \{\text{leapfrog (18)}\} \\
& gx((\bar{g}y \sqcap g)gx)^{\omega}(\bar{g}y \sqcap g)\bar{g} \\
\sqsubseteq\ & \{\text{isotony, guards form a Boolean algebra}\} \\
& gx(gx)^{\omega}\bar{g}y\bar{g} \\
=\ & \{\text{preservation assumption}\} \\
& gx(gx)^{\omega}\bar{g}y,
\end{aligned}
$$

which by isotony and idempotency of demonic choice establishes what we wanted to prove (this direction is similar to that of Kozen [10]).

If y does not preserve g, then we can introduce a new test f that is preserved by y and a program z that sets f to g. We can then insert the program $z;\ (f \leftrightarrow g)$ before the loop and into the loop body. This means that the programs

$$z;\ (f \leftrightarrow g);\ \mathsf{while}\, g\, \mathsf{do}\, x;\ z;\ (f \leftrightarrow g)\ \mathsf{od};\ y$$

and

$$z;\ (f \leftrightarrow g);\ \mathsf{while}\, f\, \mathsf{do}\, x;\ z;\ (f \leftrightarrow g)\ \mathsf{od};\ y$$

are equivalent, and we can replace the former with the latter – for which the commutativity assumption that y preserves f holds. The proof that these two programs are indeed equivalent is left to the reader (for the weak iteration version, see Kozen [10]).

Step 4: Composition. The last step we need to consider before finishing our proof is that of composing two programs in normal form. In this step, we can almost exactly follow Kozen's proof. What we want to do is, then, to transform the program

$$x_1;\ \mathsf{while}\, g_1\, \mathsf{do}\, y_1 \mathsf{od};\ x_2;\ \mathsf{while}\, g_2\, \mathsf{do}\, y_2\ \mathsf{od} \tag{34}$$

into a program in normal form. First, by Step 3, we can move x_2 into the first while loop. The program x_1 can, as Kozen notes, be ignored as it can be included in the precomputation of the resulting normal-form program. It thus suffices to show that

$$\mathsf{while}\, g\, \mathsf{do}\, x\ \mathsf{od};\ \mathsf{while}\, h\, \mathsf{do}\, y\ \mathsf{od} \tag{35}$$

can be turned into normal form.

We can assume that g commutes with y without loss of generality, just like in Step 3. This means that g commutes with the second while loop, since

$$
\begin{aligned}
& yg \sqsubseteq gy \\
\Rightarrow\ & \{\text{isotony}\} \\
& hyg \sqsubseteq hgy
\end{aligned}
$$

$\Leftrightarrow$ {guards form a Boolean algebra}

$hyg \sqsubseteq ghy$

$\Rightarrow$ {strong iteration preserves outer commutativity, see von Wright [16]}

$(hy)^\omega g \sqsubseteq g(hy)^\omega$

$\Rightarrow$ {isotony}

$(hy)^\omega g\bar{h} \sqsubseteq g(hy)^\omega \bar{h}$

$\Leftrightarrow$ {guards form a Boolean algebra}

$(hy)^\omega \bar{h}g \sqsubseteq g(hy)^\omega \bar{h}$

holds. This, in turn, means that we can use Step 3 to turn the program into

$$\begin{array}{l} \text{if } \bar{g} \text{ then while } h \text{ do } y \text{ od} \\ \quad \text{else while } g \text{ do } x \\ \qquad\qquad \text{if } \bar{g} \text{ then while } h \text{ do } y \text{ od else skip fi} \\ \qquad\qquad \text{od.} \end{array} \tag{36}$$

We can now apply Step 1 to the inner conditional, which yields two nested while loops. These nested while loops can be transformed with Step 2. Finally, we can apply Step 1, which gives us a program in normal form.

The transformations of the steps above yield a systematic method for transforming any program into normal form by inductively moving while loops outwards, starting from the innermost loop. □

8. A normal form for action systems

Action systems comprise a formalism originally intended for reasoning about concurrent systems, but they have also been used for other purposes [1]. In a sense, action systems are a generalisation of while programs, so that the action system

$$y;\ \text{do } x_1 [] x_2 [] \ldots [] x_n \text{ od}; z$$

is an iteration of the action–system body $x_1 [] x_2 [] \ldots [] x_n$, that for each iteration executes any nondeterministically chosen action $x_1, x_2, \ldots$ or x_n and terminates only when none of the actions is enabled. The initialising action y can be thought of as setting certain variables and the finalising action z as, for example, removing local variables. It is here assumed that the initialising and the finalising actions are always enabled. The do $\cdots$ od part is referred to as the action–system loop. Similarly to while programs, action systems can be combined with each other using sequential composition and nesting (but there is no explicit conditional). In this section we discuss a normal form for action systems.

We shall say that an action system is in *normal form* if it is of the form

$$y;\ \text{do } x_1 [] x_2 [] \ldots [] x_n \text{ od},$$

where y and $x_1, x_2, \ldots, x_n$ are free from action–system loops. This matches the normal form stated above for while programs.

With the aid of the enabledness operator of Solin and von Wright [15], the first action system above can be formulated in refinement algebra as

$$y;\ (x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n)^\omega;\ \overline{\epsilon x_1};\ \overline{\epsilon x_2};\ \ldots \overline{\epsilon x_n};\ z,$$

where, for any x, the guard ϵx checks whether x is enabled or not and, thus, $\overline{\epsilon x}$ is a guard that checks whether or not x is disabled.[4] Since it is assumed that the initialising and finalising actions are always enabled, one thus has $\epsilon y = \epsilon z = 1$. Moreover, in order to formulate the normal form we need to make use of *switching programs* (this manœuvre is sometimes called "use of control variables"). A switching program can set a guard to hold from scratch, for example $a = ag$ means that a sets g to hold. Switching programs will be denoted by a, b or c and are assumed to always be enabled.

The refinement-algebraic proof of the normal form for action systems is likely to be as detailed as the one for while programs and would go beyond the intended scope of this paper. But we nevertheless state the claim below and outline a proof.

[4] For the details and the predicate-transformer semantics of the enabledness operator, consult Solin and von Wright [15] and the references therein.

Conjecture 1. *Every action system, appropriately augmented with switching programs and when reasoning under preservation assumptions of the form*

$$\begin{cases} gxg &= gx, \\ \bar{g}x\bar{g} &= \bar{g}x \end{cases}$$

is equivalent to an action system in normal form.

Outline of proof. Proceeding by induction over the structure of action systems, one needs to deal with four cases.[5]

Step 1: Composition. The action system

u; do x od; v; do y od

needs to be proved equivalent to the action system

u; a; g; do g; x [] g; $\overline{\epsilon x}$; v; b; $\bar{g}$ [] $\bar{g}$; y od,

where a sets g to hold and b sets g to not hold, and g is preserved by x, y and v. When entering the action system loop only the first action will be enabled thanks to the switching program a (if the action is enabled in the first place). When x has disabled itself (were that to happen at all), then the only action enabled is the second, which in turn allows v to be executed exactly once. Upon this only y is enabled and will be so until iteration makes it not so. It is here crucial that v is always enabled, since if v were to do magic this could not be done with the aid of the latter action system. Note also that the guards after the switching programs need not be written out explicitly, but are stated for clarity here and henceforth.

Step 2: Nesting. The action system

do u; do x od od

needs to be proved equivalent to the action system

a; g; do g; u; b; $\bar{g}$ [] $\bar{g}$; x [] $\bar{g}$; $\overline{\epsilon x}$; u od

where a sets g to hold and b sets g to not hold, and g is preserved by u and x. This guarantees that the loop starts with u, whereafter u and x can be interleaved in any fashion.

Step 3: Eliminating postcomputations. The action system

do x od; y

needs to be proved equivalent to the action system

a; g; do g; x [] g; $\overline{\epsilon x}$; y; b; $\bar{g}$ od,

where a sets g to hold and b sets g to not hold, and g is preserved by x and y. The rationale is straightforward, but note that it is essential that y is always enabled.

Step 4: Choice. As prescribed by the normal form, the body of an action system may not contain action system loops. To deal with such actions, one needs first to prove that the action system body

u; do x od [] v; do y od

is equivalent to

$(u; a; g \sqcap v; b; \bar{g})$; do gx [] $\bar{g}y$ od,

where a sets g to hold and b sets g to not hold, and g is preserved by x and y. Here the choice is made before entering the action–system loop and the fresh guards guarantee that only one of the actions in the loop will be executed. No matter if, for example, y would change the state so that x would be enabled, the guard g – which is preserved by y – still hinders x from being executed.

By Step 2 one can then show that any action–system loop consisting of a choice between n action systems can be turned into one single action–system loop. The current step (Step 4) also covers the case where one of the subactions does not contain an action–system loop, since one can rewrite any action x as x; do magic od (but one must keep in mind that x must be enabled, which means that one cannot express magic as an action system). Moreover, although the initialising action contains a demonic choice, this is not part of the syntax of action systems traditionally conceived. It is however easy to show

[5] Although the proof is here similar to the proof regarding while programs, one here needs to choose a base other than "choice" (corresponding to "conditional" in the while-program case). "Choice" is here the only step (Step 4) that actually relies on earlier steps.

that this action can be made into an action system with a choice between two actions and so the construction of Step 1 can be applied. □

Using abstract refinement algebra for filling in the details of this proof outline provides an interesting open problem for people working with pen and paper as well as for people working with automated theorem provers.

9. Conclusion

The abstract-algebraic method both initiated and revived by the work of Kozen, Cohen, von Wright, the Desharnais-Möller-Struth trio and others (see for example [4,5,9–12,15–17]) should be interesting for several different communities. As we hope to have shown in this paper, abstract algebra can be useful for proving properties of programs, and much of the work done in related frameworks can be reused, or only slightly modified, to yield interesting results. The abstract-algebraic method is thus a method worth learning if one is looking for an efficient and practical reasoning tool.

Acknowledgements

I am grateful to L.A. Meinicke for her insightful comments on this paper. Thanks also to R.J.R. Back, Jules Desharnais, E.C.R. Hehner and Bernhard Möller for valuable discussions, and to numerous reviewers for helpful suggestions. I am thankful to *Sven och Dagmar Saléns Stiftelse* for financial support.

References

[1] R.J.R. Back, R. Kurki-Suonio, Decentralisation of process nets with centralised control, in: Second ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing,, ACM, 1983, pp. 131–142.
[2] R.J.R. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Springer, 1998.
[3] C. Böhm, G. Jacopini, Flow diagram Turing machines and languages with only two formation rules, Commun. ACM 9 (5) (1966) 366–371.
[4] E. Cohen, Separation and reduction, in: R.C. Backhouse, J.N. Oliviera (Eds.), MPC 2000, LNCS, vol. 1837, Springer, 2000, pp. 45–59.
[5] J. Desharnais, B. Möller, G. Struth, Kleene algebra with domain, ACM Trans. Comput. Log. 7 (4) (2006) 798–833.
[6] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall International, 1976.
[7] D. Harel, On folk theorems, Commun. ACM 23 (7) (1980) 379–389.
[8] P. Höfner, G. Struth, Automated reasoning in Kleene algebra, in: F. Pfenning (Ed.), CADE 2007, LNCS, vol. 4603, Springer, 2007, pp. 279–294.
[9] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, Inf. Comput. 110 (2) (1994) 366–390.
[10] D. Kozen, Kleene algebra with tests, ACM Trans. Programming Lang. Syst. 19 (3) (1997) 427–443.
[11] A.K. McIver, E. Cohen, C.C. Morgan, Using probabilistic Kleene algebra for protocol verification, in: R.A. Schmidt (Ed.), RelMiCS/AKA 2006, LNCS, vol. 4136, Springer, 2006, pp. 296–310.
[12] L.A. Meinicke, K. Solin, Refinement algebra for probabilistic programs, Formal Aspects Comput. 22 (1) (2010) 3–31.
[13] G. Mirkowska, Algorithmic logic and its applications, PhD Thesis, Univ. of Warsaw, Warsaw, Poland, 1972 (in Polish).
[14] K. Solin, A while program normal form theorem in total correctness, in: R. Berghammer, A. Jaoua, B. Möller (Eds.), RelMiCS/AKA 2009, LNCS, vol. 5827, Springer, 2009, pp. 322–336.
[15] K. Solin, J. von Wright, Enabledness and termination in refinement algebra, Sci. Comput. Programming 74 (8) (2009) 654–668.
[16] J. von Wright, From Kleene algebra to refinement algebra, in: E.A. Boiten, B. Möller (Eds.), MPC 2002, LNCS, vol. 2386, Springer, 2002, pp. 233–262.
[17] J. von Wright, Towards a refinement algebra, Sci. Comput. Programming 51 (2004) 23–45.